Discussion Worksheet - Week 10

Topics: Iterators (Objects, generators, first-class functions), Prolog, Final Review!

Iterators: 1, 2, 3
Prolog: 4, 5, 6, 7, 8, 9, 10
Final Review: 11, 12, 13

1. (8 min) Haskell Hopscotch, *but in Python* [Iterators - Generators]

(Adapted from Former TA Matt Wang)

Consider the following `skip` function in Haskell, which takes an `Int` parameter `n` and returns an infinite list starting at `n` with terms `n` integers apart.

```
skip :: Int -> [Int]
skip n = [x | x <- [1..], mod x n == 0]
```

For example, `take 5 (skip 2)` returns `[2, 4, 6, 8, 10]`.

Using Python generators, implement the `skip` function. The `skip` function must not contain the keyword `return`. Also, implement the `take` function such that `take(m, skip(n))` in Python behaves identically to `take m (skip n)` in Haskell.

```
def take(n, gen):
    return [next(gen) for _ in range(n)]

def skip(n):
    i = 0
    while True:
        i += n
        yield i
```

2. (8 min) A B Cs [Iterators]

Create a Python iterator `AlphabetIterator` that generates characters in the English alphabet (lowercase a through z) cyclically. The interface for and a sample run of this iterator are shown below.
Hint: Consider the Python functions `ord` and `chr`

```
# Example Usage:
alphabet_iterator = AlphabetIterator()
iterator = iter(alphabet_iterator)

# Generate the first 10 letters in the sequence
for _ in range(10):
    print(next(iterator))

# Output (newlines compressed into spaces to save space)
# a b c d e f g h i j

# Interface
class AlphabetIterator:

    def __init__(self):
        self.current_letter =____

    def __iter__(self):

    def __next__(self):
```

```
class AlphabetIterator:

    def __init__(self):
        self.current_letter = ord(‘a’)

    def __iter__(self):
        return self

    def __next__(self):
        letter = chr(self.current_letter)
        if ord(‘a’) <= self.current_letter <= ord(‘y’):
            self.current_letter = self.current_letter % ord('z') + 1
        else:
            self.current_letter = ord(‘a’)
        return letter
```

We initialize the iterator to the number representing the Unicode character “a”. Within the __next__ function, we save the value of `self.current_letter` in letter to return after we compute what our next output should be. Since we must cycle upon reaching the letter z, we create an if statement to set the next value of `self.current_letter` to be either the next letter/Unicode character or “a” again (if we are about to pass “z”).

3. (6 min) Criss Cross Applesauce [Iterators]

Write a Python function `interleave_iter` that takes in two iterators and is a generator (i.e. returns an iterator object) of the alternating values of the iterators.

For example:

`list(interleave_iter(iter([1, 2, 3]), iter([4, 5, 6])))`

should return

`[1, 4, 2, 5, 3, 6]`

`list(interleave_iter(iter([2, 2]), iter([4, 4, 4, 4])))`

should return

`[2, 4, 2, 4, 4, 4]`

```
def interleave_iter(iter1, iter2):
   remaining_iter = None
   while True:
       try:
           yield next(iter1)
       except StopIteration:
           remaining_iter = iter2
           break

       try:
           yield next(iter2)
       except StopIteration:
           remaining_iter = iter1
           break
   while True:
       try:
           yield next(remaining_iter)
       except StopIteration:
           return
```

Note that we need the final try/except/return at the end for the `list` function to work well with the output of `iterleave_iter`.

4. (4 min) **[Prolog]**

Consider the following program written in Prolog:

```
foo("Apple").
foo(A) :- bar(A), baz(A).
```

Suppose that bar is a predicate that performs some expensive computation on A. What is one possible side effect of switching the order of the two clauses above? (consider what happens for specific inputs)

If we switch the order of the two clauses above, then we will have:

```
foo(A) :- bar(A), baz(A).
foo("Apple").
```

This means for the input “Apple”, Prolog will evaluate the foo(A) clause first, invoking bar(“Apple”). As a result, the expensive computation bar will be performed first before evaluating foo(“Apple”). In general, it is recommended to order Prolog clauses from most specific (i.e. facts like foo(“Apple”) to least specific–due to how expensive backtracking can potentially be.

5. (8 min) Palindrome Prolog Polls [Prolog]

a) (2 min) Write a rule `equal_lists` to check whether two lists contain the same elements in the same order.

```
equal_lists([],[]).
equal_lists([HeadA|TailA], [HeadB|TailB]) :-
    HeadA = HeadB,
    equal_lists(TailA, TailB).
```

b) (3 min) Write a rule `is_palindrome` in Prolog to determine whether a given input List is a palindrome (It may be helpful to use `equal_lists` from part a and/or the builtin `reverse` predicate).

```
is_palindrome(List) :-
    reverse(List, Rev),
    equal_lists(List, Rev).
```

(This can be simplified even further to just `is_palindrome(List) :- reverse(List,List))`

c) (3 min) Write a rule `make_palindrome` that takes in two input lists L1 and L2, and determines if L2 is the corresponding palindrome list for L1 (ex: if L1 = [1,2,3], L2 = [1,2,3,2,1]).

Hint: one possible solution to this problem uses the builtin append predicate

```
make_palindrome([],[]).
make_palindrome(List, Pal) :-
    reverse(List, [_|ReverseMinusFirst]),
    append(List, ReverseMinusFirst, Pal).
```

6. (8 min) Reverse esreveR [Prolog]

Consider the following partially written predicate `reverseList` which reverses a list. Identify what atoms, variables, or numbers should be written in the blanks.

```
reverseList(L1, L2) :- reverseHelper(L1, L2, __).

reverseHelper([], L2, __).
reverseHelper([X|T], L2, Acc) :- reverseHelper(__, L2, [X|__]).
```

```
reverseList(L1, L2) :- reverseHelper(L1, L2, []).   %L1

reverseHelper([], L2, L2).  %L2
reverseHelper([X|T], L2, Acc) :- reverseHelper(T, L2, [X|Acc]). %L3
```

This solution can be explained as such.

L1: This line is the main predicate. It calls reverseHelper with three arguments: the original list L1, an initially empty accumulator ([ ]), and the reversed list L2.
L2: This clause defines the base case for the reversal. When the input list is empty ([ ]), the accumulator (L2) is unified with the reversed list. This is the stopping condition for the recursion.
L3: This clause defines the recursive case. It takes the input list [X|T], and recursively calls reverseHelper with the tail of the list T, the same accumulator with X added to its head ([X|Acc]), and the final reversed list L2. The effect of this recursive call is to build up the reversed list by adding elements from the original list to the accumulator in reverse order.

7. (10 min) **[Prolog]**

a) (5 min) Tyler is trying to create a prolog predicate called `custom_list`. It takes in three arguments (one for a start number, one for an end number, one for the actual list). It returns true if the list is a list of all integers from Start to End. He has the following code, but it produces a weird output.

```
custom_numlist(Start, End, [Start]) :-
    Start =:= End.
custom_numlist(Start, End, [Start|Rest]) :-
    custom_numlist(Start + 1, End, Rest).
```

Example query:

```
?- custom_numlist(3, 8, Result).
Result = [3, 3+1, 3+1+1, 3+1+1+1, 3+1+1+1+1, 3+1+1+1+1+1].
```

Can you modify Tyler's code to get the desired output?

Solution:

```
custom_numlist(Start, End, [Start]) :-
    Start =:= End.
custom_numlist(Start, End, [Start|Rest]) :-
    Start < End,
    Next is Start + 1, % Important
    custom_numlist(Next, End, Rest).
```

Prolog will not evaluate Start + 1 in the original code and that's why we get repeated "+1" in the example query. To fix this, we can add the line "Next is Start + 1" since that will ensure we set "Next" to the integer value we desire.

b) (5 min) We now want to create the prolog rule `is_custom_permuation`, that returns true if the input list is a permutation of the list [1, 2, 3 ... ,N], where N is the size of the input list. Using your solution from part a, can you write this rule?
Hint: Prolog has a built-in `permutation` function

Example query:

```
?- is_custom_permutation([2, 4, 6, 1, 3, 5]).
true.
```

Solution:

```
is_custom_permuation(List) :-
    length(List, N),
    custom_numlist(1, N, Reference),
    permutation(Reference, List).
```

We first get the length of the list using the length function, then use our predicate from part a to create a reference list from 1 to N. Once we have that, we can simply use the permutation function and look for its validity.

8. (12 min) Shearing Sheep [Prolog]

a) (7 min) Write a Prolog predicate trim that takes in four lists, and outputs true if and only if the third and fourth lists are the first and second lists trimmed to be the length of the shorter of the two.

For example,

```
trim([1, 2, 3, 4], [5, 6], C, D).
```
should return
```
C = [1,2]
D = [5,6]
```

```
trim([1, 2, 3, 4, 5, 6], B, [1, 2, 3], [3, 3, 3]).
```
should return
```
B = [3,3,3]
```

```
trim(_, [], [], []).
trim([], _, [], []).
trim([AH|AT], [BH|BT], C, D) :-
    C = [AH| CT],
    D = [BH| DT],
    trim(AT, BT, CT, DT).
```

b) (3 min) What are the first three solutions generated when you query `trim(A, B, C, D)`? Why does it do that?

The above implementation generates:

```
1. B = []
   C = []
   D = []
2. A = []
   C = []
   D = []
3. A = [E|_]
   B = [F]
   C = [E]
   D = [F]
```

The first two match the first two facts, and the third one recursively matched the third fact, which called trim again and matched with the first fact. This makes sense because Prolog looks at the statements sequentially. Note that A and B are not generated respectively for the first two queries because I used _, which means it can be anything.

c) (2 min) While you were gone, your pet lamb, Da, got onto your computer and switched around the order of some of your prolog statements. Now, querying `trim(A, B, C, D)` results in `Fatal Error: local stack overflow`, when it didn't before. Why did this happen?

If instead, the program was written as follows:

```
trim([AH|AT], [BH|BT], C, D) :-
    C = [AH| CT],
    D = [BH| DT],
    trim(AT, BT, CT, DT).
trim(_, [], [], []).
trim([], _, [], []).
```

The above test examples would still work. However, because the recursive call is first, the unification of the first step would keep on recursing and never stop. Thus, it would cause a stack overflow error. Order matters!

9. (8 min) What is and what isn’t [Prolog]

What is the difference between "is", "=", and "==" in Prolog?

(Answer generated partially by ChatGPT)
In Prolog, “is”, “=”, and “==” serve different purposes and have distinct meanings:

“is” Operator:
The is operator is used for arithmetic evaluation. It binds the right-hand side expression to the left-hand side variable. It is specifically used when you want to perform arithmetic operations and evaluate the result.

Example:

```
X is 3 + 4.  % X is bound to the result of the arithmetic operation 3
+ 4
```

“=” Operator:
The = operator is the unification operator in Prolog. It is used to unify two terms, making them equal. It is used for both matching and assignment. If the terms on both sides can be made equal, the unification succeeds and any variables involved are instantiated to make the terms equal.

Examples:

```
X = 3 + 4.    % Unifies X with the term 3 + 4
Y = 7,        % Unifies Y with the value 7
List = [1, 2, 3].  % Unifies List with the list [1, 2, 3]
```

“==” Operator:
The == operator is the equality operator in Prolog. It checks if two terms are structurally identical. It does not perform arithmetic evaluation but rather checks whether two terms are syntactically the same.

Examples:

```
3 + 4 == 7.     % True, as both sides represent the same arithmetic
expression
X == Y.         % True if X and Y are already instantiated to the same
value
[1, 2, 3] == [1, 2, 3].   % True, both sides represent the same list
```

In summary:
Use “is” for arithmetic evaluation.
Use “=” for unification, which can be both for matching and assignment.
Use “==” for checking structural equality of terms.

10. (15 min) **[Prolog]**

A student currently taking the compilers course at UCLA, CS 132, is working on the second project of the class. The first project involved implementing a simple parser (with many limitations). The goal of the second project is simple: to write a type checker for a subset of Java called MiniJava. The class requires the type checker to be written in Java, but this student swore never to touch that language again after finishing CS 131 with Eggert.

a) (5 min) The student thought Prolog was really cool when they learned about it, so they decided to implement the project with it. Is this feasible? Can you think of any potential limitations they would run into?

This is definitely feasible! One of prolog's main applications is theorem proving and type checking. As we saw in lecture, it's used for the Java compiler's type checking system!

One limitation is that it may be difficult to implement type inference (via Java's var keyword), but it is certainly doable.

b) (10 min) Bonus: Write a type checker in Prolog that supports two types: int and bool. Your type checker should support the following operations:

- int + int → int
- int - int → int
- int * int → int
- int / int → int
- and(bool, bool) → bool
- or(bool, bool) → bool
- not_(bool) → bool

Solution (generated partially by ChatGPT):

```
% Define the types of some variables
type('x', int).
type('y', bool).

% Define logical operators
and(_, _).
or(_, _).
not_(_).

% Define the type rules for arithmetic and logical expressions
type_expr(E1 + E2, int) :- type_expr(E1, int), type_expr(E2, int).
type_expr(E1 - E2, int) :- type_expr(E1, int), type_expr(E2, int).
type_expr(E1 * E2, int) :- type_expr(E1, int), type_expr(E2, int).
type_expr(E1 / E2, int) :- type_expr(E1, int), type_expr(E2, int).
type_expr(and(E1, E2), bool) :- type_expr(E1, bool), type_expr(E2,
bool).
type_expr(or(E1, E2), bool) :- type_expr(E1, bool), type_expr(E2,
bool).
type_expr(not_(E), bool) :- type_expr(E, bool).

% Define the type rule for variables
type_expr(E, T) :- atom(E), type(E, T).
```

11. (8 min) **[Final Review - Parametric Polymorphism]**

Consider two identical programs, except one uses templates and the other uses bounded generics. With specific reference to static and dynamic dispatch, determine which one will take less time *at runtime* to execute.

The key difference to consider is that templates will create duplicate versions of a function/class at compile time whereas a compiled generic will reuse a single version. While this leads to additional compile-time overhead in the templated case, it enables the use of static dispatch at runtime–since every type will have its own function, the compiler does not need to determine which function to call at runtime. With generics, the runtime determination of an object's type is necessary to dispatch to the correct method implementation.

12. (6 min) **[Final Review - Parametric Polymorphism]**

a) (3 min) Between templates and generics, is one approach “more conservative” than the other, i.e. are there programs that compile for templates that don’t compile for generics, or vice versa?

Generics are more conservative, since they will fail on valid code if function types are not properly bounded (and you perform a type-specific operation). Consider the following C++ code:

```
<typename T>
void foo (T t) {
    t.quack();
}
foo(new Duck()); // Assuming Duck::quack exists!
```

The above is technically a valid program, but fails to compile with generics because we didn’t specify that the type `T` implements the `quack` function.

b) (3 min) How can you tell the difference between a language that implements parametric polymorphism using generics vs. using templates?

If the language calls functions on/of a type without first bounding the type (and doesn’t error), then it’s implemented using templates. If there’s code bounding the input types to a function, then it’s likely generics. Otherwise, there may be ambiguity for which it actually is.

13. (6 Min) Handling Errors Swiftly [Final Review - Error Handling]

Consider this dummy API to fetch a User, written in Swift.

```swift
struct User {
    let username: String
    let age: Int
}

enum APIError: Error {
    case requestFailed
    case invalidData
}

func getUserFromAPI() -> InsertReturnTypeHere {
    // Simulate a successful request sometimes
    if Bool.random() && Bool.random() { // Line A
        let user = User(username: "JohnDoe", age: 25)
        return .success(user)
    } else {
        return .failure(.requestFailed)
    }
}

// Example usage:

switch getUserFromAPI() {
case .success(let user):
    print("User fetched successfully: \(user.username), \(user.age)
years old.")
case .failure(let error):
    switch error {
    case .requestFailed:
        print("Failed to fetch user. Please try again.")
    case .invalidData:
        print("Received invalid data from the server.")
    }
}
```

a) (3 min) What type of error handling is Swift using here?

This is an example usage of **result objects**. Here, the function `getUserFromAPI()` returns a result object containing either a valid result (a `User` object containing the information of some “John Doe”) or an Error indicating that the request has failed.

b) (3 min) Assuming Line A evaluates to true, what will our program output? What if it evaluates to false?

```
func getUserFromAPI() -> InsertReturnTypeHere {
    if Bool.random() && Bool.random() // Line A
      ...
}
```

Line A evaluates to true → program outputs:
`User fetched successfully: JohnDoe, 25 years old.`

Line A evaluates to false → program outputs:
`Failed to fetch user. Please try again.`